Leibniz Universität Hannover

Programmieren 1: Reguläre Ausdrücke, Lambda-Ausdrücke

Human-Computer Interaction Group

Prof. Dr. Michael Rohs
michael.rohs@hci.uni-hannover.de

Vorlesungen

Termin	Datum	Thema	Literatur
1	17.10.	Organisatorisches, Algorithmen, Beschreibungsebenen	M 1
2	24.10.	C Sprachelemente, Compiler, Ein- & Ausgabe	K 1 & 7
3	31.10.	Datentypen, Wertebereiche, Ausdrücke, Operatoren	K 2; M 5
4	7.11.	C Kontrollstrukturen, externe Variablen, Funktionen	K 3 & 4
5	14.11.	C Programmstruktur, Rekursion, Strukturen	K 4-6
6	21.11.	Zeiger, Zeiger und Felder, Listen	K 5 & 6
7	28.11.	Anwendung von Listen, Dynamischer Speicher	K 5 & 6
8	5.12.	Funktionszeiger, Makefiles, Abschluss C	K 5-7
9	12.12.	Java Programmstruktur, Kontrollstrukturen, Methoden	M 2-4, 6
10	19.12.	Java Arrays, Zeichen und Strings in Java	M 7-9
	26.12.	frei	
	2.1.	frei	
11	9.1.	Reguläre Ausdrücke, Lambda-Ausdrücke	M 10 & 11
12	16.1.	Objektorientierung	M 11 & 12
13	23.1.	Dynamische Datenstrukturen	M 13 & 14
14	30.1.	Vererbung, Aufzählungen, Zusammenfassung	

Sprechstunde

- Wann?
  - Montags 10-12 Uhr
- Wo?
  - FG Mensch-Computer-Interaktion
  - Appelstr. 9A
  - 9. Etage
  - Raum 906 (Klingel am Etageneingang)

Review

- Classes (a second glimpse)
- Arrays (1D, 2D, nD)
- Characters, Unicode
- Strings, String operations
- Edit distance

Preview

- Regular expressions
- Lambda expressions

Review: Objects, Object Comparison

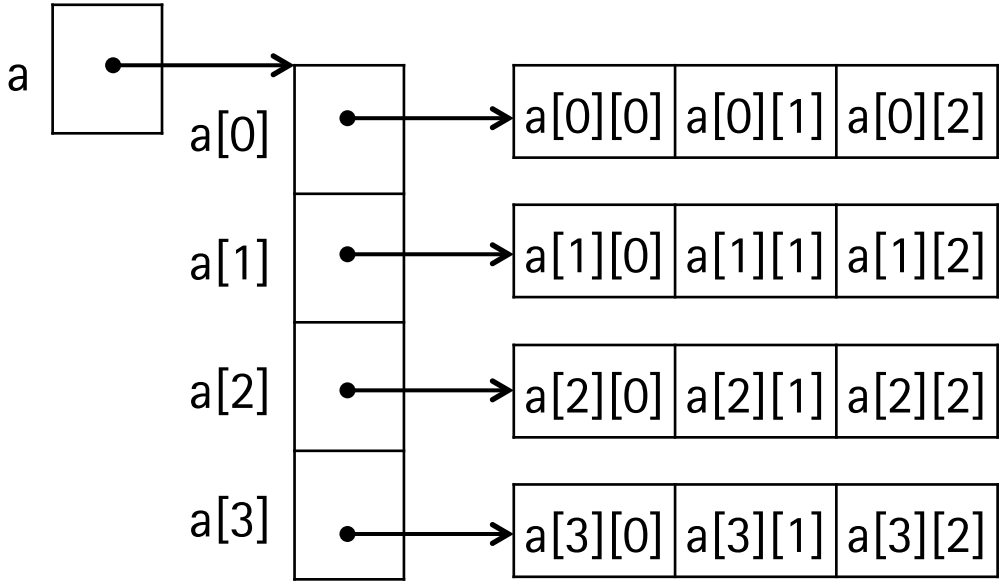
- Given class Date →
  - What is Date(int d, int m, int y) ?
- What happens for:
  - Date d1 = new Date(1,1,2015);
  - Date d2 = new Date(9,1,2015);
  - d2.day++;
  - if (d1 == d2) …
  - d1.day = d2.day; if (d1 == d2) …
  - if (d1.isEqual(d2))…
  - d1 = d2;
  - if (d1 == d2) …

```
class Date {
    int day;
    int month;
    int year;
    Date(int d, int m, int y) {
        day = d; month = m; year = y;
    }
    boolean isEqual(Date d) {
        return day == d.day &&
          month == d.month &&
          year == d.year;
    }
}
```

Review: Two-Dimensional Arrays

- What happens for:
  - int[][] a = new int[4][3];
  - Out.println(a.length);
  - Out.println(a[0].length);
- Memory layout?

a[0][0]	a[0][1]	a[0][2]
a[1][0]	a[1][1]	a[1][2]
a[2][0]	a[2][1]	a[2][2]
a[3][0]	a[3][1]	a[3][2]



Review: Strings

- Strings are immutable.
  - What does that mean? Why?
- Difference between String and StringBuilder?

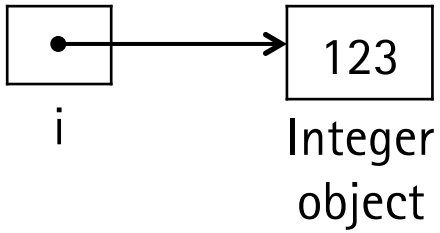
- How to compare strings?
  - String s = In.readString();
  - if (s.equals("Hello")) ...

- Many useful String methods, see JDK Javadoc
  - http://docs.oracle.com/javase/8/docs/api/java/lang/String.html
  - http://docs.oracle.com/javase/8/docs/api/index.html
  - http://docs.oracle.com/javase/8/

Review: Primitive Types and Wrapper Classes

- Difference between Integer, Long, Float, Double and int, long, float, double?
  - Why needed?
- Conversion between primitive types and wrapper classes?
  - int to Integer: valueOf(int i) (static factory method)
  - Integer to int: int intValue() (instance method)
- Conversion to/from class String?
  - int i = Integer.parseInt("123");
  - float f = Float.parseFloat("3.14");
  - String s = String.valueOf(f);
  - char[] a = s.toCharArray();

Integer i = Integer.valueOf(123);

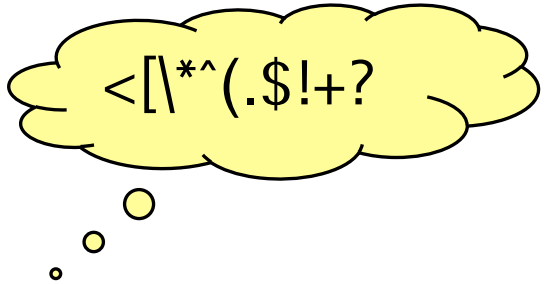


Review: Primitive Types and Wrapper Classes

- Implementation of Wrapper class Integer?
  - Data, constructor(int a), intValue(), valueOf(int i)

```
class Integer {
    int i;
    Integer(int a) { i = a; }
    int intValue() { return i; }
    static Integer valueOf(int i) {
        return new Integer(i);
    }
}
```

REGULAR EXPRESSIONS



Regular Expressions

- A "language" for describing sets of strings
  - Separate from Java
- Useful for pattern matching in strings
- Used in text editors, text processing, command line tools, URL parsing, compilers, etc.
  - Find all URLs on a Web page
  - Verify input in Web forms

- Parsers for regular expressions can be implemented with finite state machines
  - Parser for regular expression "accepts" input string if in accepting state at end of the string

Verifying Email Address Format

- Example: Verifying email addresses of the form <first>_<last>@<server>.de
  - Check for underscore ('_')
  - Check for '@'
  - Check for ".de" at the end
  - Check for letters before '_' and between '_' and '@'
- Define a regular expression pattern

```
String regex = "[a-z]+_[a-z]+@[a-z]+\\.de";

String email = "james_bond@test.de";
if (email.matches(regex)) {
      System.out.println("email format correct");
}
```

Pattern describes allowed format

Test whether email address matches format

Email Regular Expression Decomposed

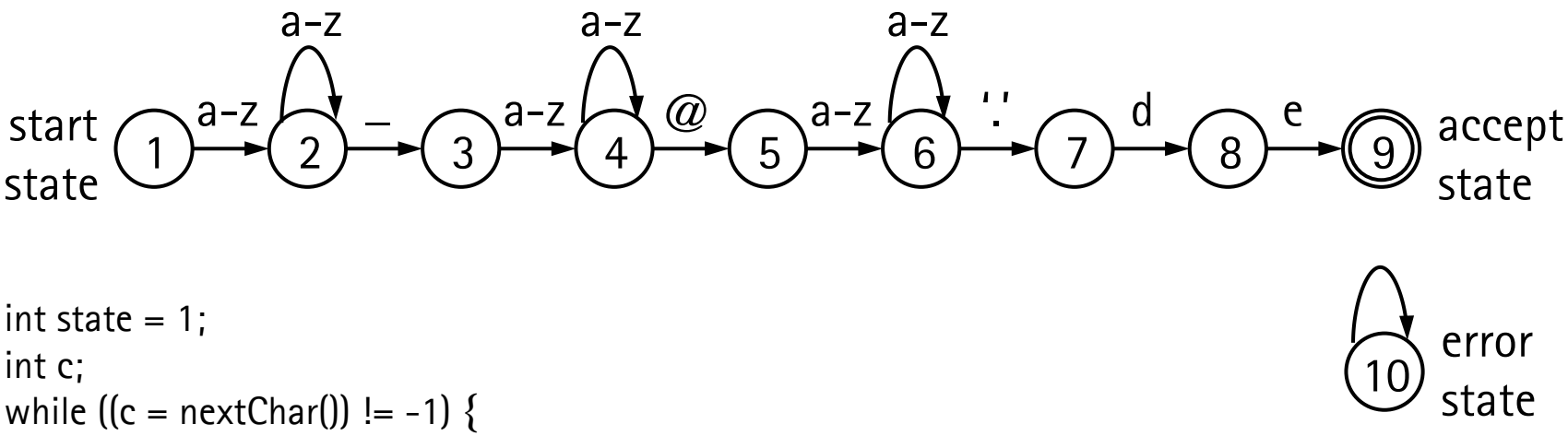
- String regex = "[a-z]+_[a-z]+@[a-z]+\\.de";
- [a-z] one of the characters in [ ] (here: lower-case letters)
- [a-z]+ one or more of the characters [ ]
- _ underscore character
- @ the '@' character
- \\. the '.' character

- Backslash is escape character in Java String and in reg. exp.
  - . matches any character
  - \. matches the dot character (except in [ ])
  - Backslash needs to be escaped in Java String literal: "\\."

Implementing Regular Expressions

String regex = "[a-z]+_[a-z]+@[a-z]+\\.de";

Finite state machine:




```
int state = 1;
int c;
while ((c = nextChar()) != -1) {
      if (state == 1) {
            if (c >= 'a' && c <= 'z') state = 2;
            else state = 10;
      } else if (state == 2) ...
}
```

Basic Regular Expression Syntax

Character	Meaning	Example	Example matches
*	zero or more	a*	"", a, aa, aaa, …
+	one ore more	a+	a, aa, aaa, …
.	any character	a.a	aaa, aba, a3a, aXa, …
?	optional	colou?r	color, colour
[…]	one of these	[hc]at	hat, cat
[…]	not any of these	a[^bc]d	a3d, axd, **but not** abd, acd
…|…	alternative	in|on	in, on
(…)	groups	(i|o)n	in, on
^	start of string	^def	def, only at start of string/line
$	end of string	C$	C, only at end of string/line

* and + are *quantifiers*

[…] is a *character class*

^ and $ are *boundary matchers*

String regex = "[A-Za-z]+_[A-Za-z]+@[A-Za-z]+\\.de";

Method matches in class String

- matches checks, if complete string matches regular expression

```
if (s.matches("[0-9][0-9]?[/.][0-9][0-9]?[/.][0-9][0-9][0-9]?[0-9]?"))
        Out.println("match");
```

- Example matches:

```
String s = "09.01.2015";
String s = "09.1.2015";
String s = "09.1.15";
String s = "9/1/15";
```

- Dot . character
  - Normally . stands for “any character”
  - In [.] the dot stands for itself

Ranges

- System.out.println("-".matches("[a-z]"));
  - false
- System.out.println("-".matches("[az-]"));
  - true
- System.out.println("b".matches("[a-z]"));
  - true
- System.out.println("b".matches("[az-]"));
  - false

Regular Expression API in Java

- String class
  - matches(...) to check if a string matches a pattern
  - replaceAll(...) and replaceFirst(...) to replace matched part
  - split(...) to split a string into an array of strings
- Pattern class
  - A compiled regular expression
- Matcher class
  - Matching a string with a pattern, maintains state

http://docs.oracle.com/javase/8/docs/api/
http://docs.oracle.com/javase/tutorial/essential/regex/

Groups in Regular Expressions

- Substrings can be treated as groups using (...)
- Groups can be referenced later by number
- Example: Time

```
String time = "09:15";
Pattern p = Pattern.compile(    "([0-2][0-9]):([0-5][0-9])");
Matcher m = p.matcher(time);
if (m.matches()) {
    System.out.println("hour = " + m.group(1) + ", minute = " + m.group(2));
}
```

group 1 group 2

group 1 matched "09"

group 2 matched "15"

- Output

```
hour = 09, minute = 15
```

Pattern Class

- Represents a regular expression
  - Needs at top of file: import java.util.regex.Pattern;
- compile - produce a compiled pattern (i.e., a finite state machine)
  - Pattern p = Pattern.compile("([0-2][0-9]):([0-5][0-9])");
- matcher - create a Matcher object
  - Matcher m = p.matcher("09:15");

Matcher Class

- Performs regular expression matching given a pattern and an input string, maintains current matching state
  - Needs at top of file: import java.util.regex.*;

- boolean matches() - check if entire input matches the pattern
- boolean find() - look for next substring that matches the pattern
- int start() - start index of match (need to call find() before)
- int end() - index of last character matched + 1
- String group() - matched substring
- String group(int i) - the i-th group of the previous match
- boolean hitEnd() - end of input is reached

Example: Extracting hrefs from an HTML File

```
import java.util.regex.*;
class UrlsFromHtml {
    static void getURLs() {
        In.open("luh.html");
        String html = In.readFile();
        In.close();

        Pattern p = Pattern.compile("href=\"([^\"]+)\"");

        Matcher m = p.matcher(html);
        while (m.find()) {
            System.out.println(m.group(1));
        }
    }

    public static void main(String[] args) {
        getURLs();
    }
}
```

HTML:

```
<a href="http://www.uni-hannover.de/">
Leibniz Universität Hannover</a>
```

Group References

- Reference a group within the same regular expression
- Example: Date format that allows either "." or "/" as separator
  - "10.1.2015" ✔, "10/1/2015" ✔, "10.1/2015" ✖, "10/1.2015" ✖
  - `"[0-9][0-9]?([/.])[0-9][0-9]?\\1[0-9][0-9][0-9]?[0-9]?"`

More Regular Expressions Syntax

Pattern	Meaning	Definition/Example
\d	digit	[0-9]
\D	non-digit	[^0-9]
\w	word character	[A-Za-z_0-9]
\W	non-word character	[^\w]
\s	whitespace character	[\t\n\r]
\S	non-whitespace character	[^\s]
\b	word boundary	
{n}	character repeated n times	x{3}: xxx
{n,m}	character repeated n to m times	x{1,3}: x, xx, xxx

Umlaute: use Pattern.compile(pt, Pattern.UNICODE_CHARACTER_CLASS);

Further Examples

1. German Postleitzahl

   `"\\d{5}"`

2. Host part of a http-URL

   `"http://([a-z0-9.-]+)"` → group(1)

3. A 6-10 character password of only digits or lower-case letters, starting with a letter, ending with a digit

   `"[a-z][a-z0-9]{4,8}\\d”`

Named Groups

- Groups are numbered consecutively
- Reference by number can lead to errors

- Named groups
  - `"(?<myDigits>\\d+)"`

- Reference by name

```
if (matcher.find()) {
      String myDigits = matcher.group("myDigits");
}
```

Greedy and Reluctant Quantifiers

- Greedy quantifiers: x?, x*, x+, x{n}, x{n,m}
  - They try to match as much as possible (they are "greedy")
- Example
  - Input: "The new year 2015"
  - Pattern: "`.*([0-9]+)`"
  - Group 1: "5" because greedy ".*" captures "The new year 201"
- Reluctant quantifiers: x??, x*?, x+?, x{n}?, x{n,m}?
  - They try to match as little as possible (they are "reluctant")
- Example
  - Input: "The new year 2015"
  - Pattern: "`.*?([0-9]+)`"
  - Group 1: "2015" because reluctant ".*?" captures "The new year "

LAMBDA EXPRESSIONS

Java 8!

Reacting to a Button Press

- A lambda expression to react to a button press

```
button.setAction(event -> System.out.println("Thanks for clicking!"));
```

lambda expression

- Do not call immediately, but only when button is pressed

```
button.setAction(event -> System.out.println("Thanks for clicking!"));
```

parameter(s) function body

- Lambda expression: Anonymous function
  - Parameters, function body, free variables (none in this example)
  - Can be stored like data for later execution

Lambda Expressions

- An expression with parameter variables (and free variables)
  - (int x) -> { return 2 * x + 1; } or x -> 2*x + 1
- Why lambda?
  - In the 1930s Alonzo Church tried to formalize computability using function abstraction, function application, and variable binding
  - Principia Mathematica (Whitehead, Russel, 1910-13) used $2\hat{x}+1$ to denote function f with f(x) = 2x+1. Alonzo Church wanted the notation $\hat{x}.2x+1$, the typesetter could only do ^x.2x+1 (uppercase lambda), another typesetter changed this to λ x.2x+1 (lowercase lambda)
  - λ x.2x+1 is called a lambda abstraction for the function f(x) = 2x+1
- Function application in lambda calculus
  - (λ x . 2x+1) 3 = 2*3+1 = 7
  - (λ op .(λ x .op x x)) (+) 3 = (λ x .(+) x x) 3 = (+) 3 3 = 6

Functions as Values / Code as Data

- Functions as values
  - Store in variables, pass as arguments to other functions, return from functions
- Remember C function pointers
  - Function pointer is memory address of machine code of a function
- Why functions as values?
  - Callbacks in GUIs (button clicked)
  - Separating iteration and operation on elements (filter elements from list)
  - Comparator functions for sorting (case-insensitive string sorting)
  - Parallel operations on big data (function describes operation on each element)
  - Allows for very general and concise code

Things – Actions

Things

Nouns

Data

Actions

Verbs

Procedures

Knock down this wall...

Brian Harvey, MIT CS 61A, The Structure and Interpretation of Computer Programs, Lecture 3: Functions of Functions

Lambda Expressions in Java

(int x, int y) -> { System.out.println(x + y); }
(x, y) -> System.out.println(x + y)
(int x, int y) -> { return x+y; }
(x, y) -> x + y
x -> 2 * x

- Syntax:
  (formal-parameter-list) -> { expression-or-statements }
- (formal-parameter-list)
  - (int x, int y)
  - (x, y) may omit variable type, will be inferred
  - x single parameter, may omit parentheses
  - ( ) empty parameter list
- {expression-or-statements}
  - { return x+y; } statements must be enclosed in braces
  - x+y single expression or method call, may omit parentheses
  - { System.out.println(x+y); }

Interfaces in Java

- Interfaces specify method signatures without implementing them
  - (Java 8 allows implementing static and default methods)
- The set of method signatures of an interface specify a type
- Interfaces specify what is provided by a component
- Objects can later implement these interfaces

- Example interface

```
interface Animal {
    void eat(String food);
    String speak();
    boolean canFly();
}
```

- Example class implementing interface

```
class Cat implements Animal {
    public void eat(String food) { // eating... }
    public String speak() { return "miau"; }
    public boolean canFly() { return false; };
}
```

Functional Interfaces to Specify Lambda Expressions

- Lambda expressions do not have explicit type
  - Compiler infers target type from context
  - Target type is type of object to which lambda expression is bound
  - Target type is a functional interface
- A functional interface has a single abstract method
- Example

```
interface DoubleToDouble {
    double apply(double x);   // takes a double, returns a double
}
```

- Assigning a lambda expression to this type

```
DoubleToDouble square = x -> x * x;
```

- Applying the lambda expression to 3

```
double d = square.apply(3);
```

Manipulating Sentences

```
interface StringToBool { // functional interface
    boolean apply(String s);   // takes a string, returns a boolean
}
String keep(StringToBool predicate, String s) { // keep words that match predicate
    String[] words = s.split("[ ,.!;]+"); // split the sentence at spaces, commas, etc.
    StringBuilder sb = new StringBuilder();
    for (String w : words) {
        if (predicate.apply(w)) { // append w if predicate is true for the w
            sb.append(w); sb.append(' ');
        }
    }
    return sb.toString();
}
```

Manipulating Sentences

```
String t = keep(s -> s.contains("s"), "this is how you see it");
// returns "this is see "

String t = keep(s -> s.charAt(0) == s.charAt(s.length() - 1), "radar rain tilt door");
// returns "radar tilt "

String t = keep(s -> isNumber(s), "the 1 after 2 is 3!");
// returns "1 2 3 "

boolean isNumber(String word) {
    for (int i = 0; i < word.length(); i++)
        if (!Character.isDigit(word.charAt(i))) return false;
    return true;
}
```

Manipulating Sentences

```
Strint t = keep(s -> isPronoun(s), "this is how you see it");
// returns "you it "

String[] pronouns = { "I", "me", "you", "he", "she", "it", "him", "her", "we", "us" };

boolean isPronoun(String word) {
    for (String p : pronouns) {
        if (p.equals(word)) return true;
    }
    return false;
}
```

Method and Constructor References

- Instead of writing
  String t = keep(s -> isPronoun(s), "this is how you see it");
  - Lambda expression only forwards argument to desired function

- better write
  String t = keep(this::isPronoun, "this is how you see it");
  - A reference to an instance method

- Method references to static methods (ClassName::staticMethod)
  - DoubleToDouble f = Math::sin; // static method sin
  - double x = f.apply(1.1);

Method and Constructor References

- Method references to instance methods (ClassName::instMethod)
  - interface StringToInt {
    int apply(String s); // take String, return int
    }
  - StringToInt len = String::length; // instance method of class String
  - int n = len.apply("hello"); // object passed as first argument

- Constructor references (ClassName::new)
  - interface IntToInteger {
    Integer apply(int i); // take (primitive) int, return Integer object
    }
  - IntToInteger iti = Integer::new; // constructor of class Integer
  - Integer k = iti.apply(123);

Variables from Enclosing Scope, Closure

- Lambda expression can capture the value of a variable in the enclosing scope

```
final int minLength = 4; // final (constant) variable in enclosing scope
String t = keep(s -> s.length() >= minLength, "this is how you see it");
// returns "this "
```

- Free variables: Not parameters and not defined in lambda body
- Can only capture (effectively) final local variables and any object and class variables
  - Effectively final: Variable that does not change (after initialization)
- Lambda expressions do not introduce a new level of scoping

Effectively Final Local Variables, Closure

- Effectively final local variables

```
String f(String t) {
    int minLength = 4; // effectively final (not modified after initialization)
    t = keep(s -> s.length() >= minLength, t);
    // minLength++;    // uncommenting this yields a compiler error
    return t;
}
```

- A closure is the combination of
  - Parameter variables (s in example above)
  - Function body (code in {...} in example above)
  - Values for free variables (4 in example above)

Effectively Final Local Variables, Closure

- Return a function from a function

```
StringToBool makePredicate(int minLength) {
    return s -> s.length() >= minLength;
}
```

- Returned StringToBool-function can be used after makePredicate has completed!

```
StringToBool f = makePredicate(4);
System.out.println(f.apply("1234")); // returns true
System.out.println(f.apply("123")); // returns false
```

Processing an Array of Persons

```
class Person {
    String name;
    int age;
    String emailAddress;
    Person(String name, int age, String emailAddress) { // constructor
        this.name = name;
        this.age = age;
        this.emailAddress = emailAddress;
    }
    public String toString() {
        return "Person(" + name + ", " + age + ", " + emailAddress + ")";
    }
}
```

Processing an Array of Persons

- An array of persons

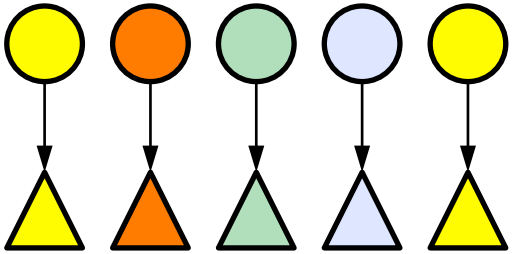
```
Person[] ps = {
    new Person("Karl", 23, "karl@abc.de"),
    new Person("Franz", 29, "franz@def.de"),
    new Person("Alice", 19, "alice@jkl.de"),
    new Person("Fritz", 25, "fritz@ghi.de"),
    new Person("Alina", 35, "alina@abc.de")
};
// print the array
for (Person p : ps) {
    System.out.println(p.toString());
}
```

- Output

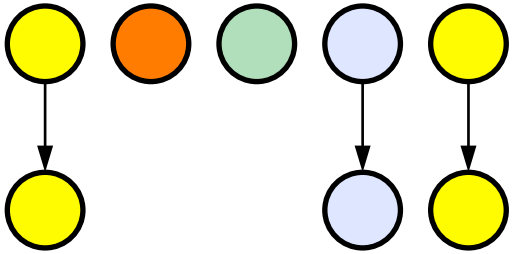
```
Person(Karl, 23, karl@abc.de)
Person(Franz, 29, franz@def.de)
Person(Alice, 19, alice@jkl.de)
Person(Fritz, 25, fritz@ghi.de)
Person(Alina, 35, alina@abc.de)
```

Reminder: map, filter, reduce

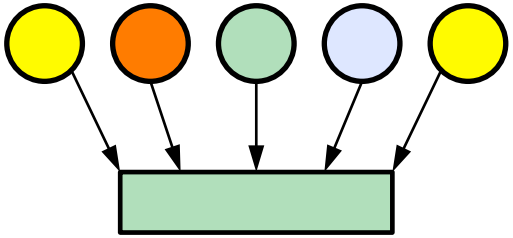
map:



filter:



reduce:



Processing an Array of Persons

- Only persons aged 20-25

```
Person[] ps2 = filterPersons(p -> p.age >= 20 && p.age <= 25, ps);
```

- Need their email addresses

```
String[] ps3 = mapPersonToString(p -> p.emailAddress, ps2);
```

- Functional interface for predicate in filterPersons

```
interface PersonToBoolean {
    boolean apply(Person p); // in: Person, out: boolean
}
```

- Functional interface for transformation in mapPersonToString

```
interface PersonToString {
    String apply(Person p); // in: Person, out: String
}
```

Filtering an Array of Persons

```
Person[] filterPersons(PersonToBoolean predicate, Person[] ps) {
    int n = 0;
    for (Person p : ps)
        if (predicate.apply(p)) // first, count how many persons pass the test
            n++;
    Person[] result = new Person[n]; // result array for n persons
    int i = 0;
    for (Person p : ps)
        if (predicate.apply(p)) // add to result if passes test
            result[i++] = p;
    return result;
}
```

How to avoid calling this a second time?

Predicate (test) interface:

```
interface PersonToBoolean {
    boolean apply(Person p);
}
```

Mapping an Array of Persons to an Array of Strings

```
String[] mapPersonToString(PersonToString f, Person[] ps) {
    String[] result = new String[ps.length];
    int i = 0;
    for (Person p : ps) {
        result[i++] = f.apply(p); // apply transformation function to each element
    }
    return result;
}
```

Transformation interface:

```
interface PersonToString {
    String apply(Person p);
}
```

Generalization

- Tedious to write filter and map functions for each class
  - filterPerson, filterCat, etc.
  - mapPersonToString, mapCatToString, etc.
- Can write more general functions that process any Object
  - filterObject, mapObjectToObject
  - Need a list instead of arrays to do that
- In Java, an instance of any class (e.g. Persons) is also an instance of class Object
  → More on this later (inheritance)
- Need to explicitly cast Object to specific type
  - Remember void* in C

A List of Objects

```
class Node { // the linked list nodes
    Object value;
    Node next;
    Node(Object value, Node next) {
        this.value = value;
        this.next = next;
    }
}

class List { // the list itself
    Node first = null;
    Node last = null;
    ...
```

A List of Objects

```
class List {
    ...
    void append(Object value) { // append an element to the list
        if (last == null) { // empty list
            first = new Node(value, null);
            last = first;
        } else { // non-empty list
            last.next = new Node(value, null);
            last = last.next;
        }
    }
    ...
```

A List of Objects

```
class List {
    ...
    static List toList(Object[] os) { // produce a list from an array,
                                      // static factory method
        List result = new List();
        for (Object o : os) {
            result.append(o);
        }
        return result;
    }
    ...
```

A List of Objects

```
class List {
    ...
    void forEach(ObjectToVoid f) { // apply f to each list element
        for (Node node = first; node != null; node = node.next) {
            f.apply(node.value);
        }
    }
    ...
```

Type of function to apply:

```
interface ObjectToVoid {
    void apply(Object o);
}
```

A List of Objects

```
class List {
    ...
    List filter(ObjectToBoolean predicate) { // filter with predicate function
        List result = new List();
        for (Node node = first; node != null; node = node.next) {
            if (predicate.apply(node.value)) {
                result.append(node.value);
            }
        }
        return result;
    }
    ...
```

Type of predicate to apply:

```
interface ObjectToBoolean {
    boolean apply(Object o);
}
```

A List of Objects

```
class List {
    ...
    List map(ObjectToObject f) { // map with transformation function
        List result = new List();
        for (Node node = first; node != null; node = node.next) {
            result.append(f.apply(node.value));
        }
        return result;
    }
    ...
```

Type of transformation to apply:

```
interface ObjectToObject {
    Object apply(Object o);
}
```

Using the List of Objects

```
Person[] ps = { new Person("Karl", 23, "karl@abc.de"), ... };
List list = List.toList(ps); // produce list from array
list.forEach(System.out::println); // apply print function to each element

// only persons aged 20-25
List ps2 = list.filter(p -> ((Person)p).age >= 20 && ((Person)p).age <= 25);
ps2.forEach(System.out::println); // print each element

// need their email addresses
List ps3 = ps2.map(p -> ((Person)p).emailAddress);
ps3.forEach(System.out::println); // print each element

List list = List.toList(ps).filter(p -> ...).map(p -> ...) // all at once
```

Explicit Type Casts from Object to Concrete Type

```
List ps2 = list.filter(p -> ((Person)p).age >= 20 && ((Person)p).age <= 25);
```

ObjectToBoolean function

```
List ps3 = ps2.map(p -> ((Person)p).emailAddress);
```

ObjectToObject function

- Type of p is Object, because
  - filter takes ObjectToBoolean function
  - map takes ObjectToObject function
- Type cast from Object to Person: ((Person)p)
- Type cast does not change the object/person, just the reference type

Summary

- Regular expressions
  - Enable specifying patterns in strings
  - Simple example: Split sentence into words
    ```
    String[] words = sentence.split("[ ,.!;]+");
    ```
- Lambda expressions
  - Enable treating code as data
  - Simple example: Register button click handler
    ```
    button.setAction(event -> System.out.println("Thanks for clicking!"));
    ```